Available online at www.sciencedirect.com

SciVerse ScienceDirect

Procedia Computer Science 9 (2012) 57 – 66

Procedia Computer Science

International Conference on Computational Science, ICCS 2012

The Sliced COO format for Sparse Matrix-Vector Multiplication on CUDA-enabled GPUs

Hoang-Vu Dang, Bertil Schmidt

Institut für Informatik, Johannes Gutenberg Universität, Staudingerweg 9, 55128 Mainz, Germany

Abstract

Existing formats for Sparse Matrix-Vector Multiplication (SpMV) on the GPU are outperforming their corresponding implementations on multi-core CPUs. In this paper, we present a new format called Sliced COO (SCOO) and an efficient CUDA implementation to perform SpMV on the GPU. While previous work shows experiments on small to medium-sized sparse matrices, we perform evaluations on large sparse matrices. We compared SCOO performance to existing formats of the NVIDIA Cusp library. Our resutls on a Fermi GPU show that SCOO outperforms the COO and CSR format for all tested matrices and the HYB format for all tested unstructured matrices. Furthermore, comparison to a Sandy-Bridge CPU shows that SCOO on a Fermi GPU outperforms the multi-threaded CSR implementation of the Intel MKL Library on an i7-2700K by a factor between 5.5 and 18.

Keywords: SpMV, CUDA, Fermi

1. Introduction

Sparse Matrix-Vector Multiplication (SpMV) is a crucial operation in scientific computing. The need to accelerate this operation comes from its application in Krylov methods on large sparse matrices, whose SpMVs are performed interatively, i.e. a sequence of vectors $y_i = A^i x$ is computed. Initial work on accelerating SpMV on CUDA-enabled GPUs was published in [1]. The corresponding implementation in the Cusp library [2] include optimized implementation of the well-known Compressed Sparse Row (CSR), Coodinate List (COO), Ellpack (ELL), Hybrid (HYB) and Diagonal (DIA) formats. [1] reports speedup between 1.56 and 12.30 compared to an optimized CPU implementation for a range of sparse matrices.

Because SpMV is a largely memory bandwidth-bound operation, these speedups are mainly due to the fast GPU memory bandwidth. Reported results indicate that different access patterns to the matrix and vectors influence GPU performance. Choi at al. [3] proposed blocked versions of CSR (BCSR) and ELL (BELLPACK) which exploit block structures in some sparse matrices. Monakov at al. [4] proposed a sliced version of ELL (SLE) with either fixed or variable slice sizes that helps reducing the memory overhead of ELL by dividing the matrix into slices. These formats improve the performance compared to Cusp a factor of around 1.8 on avarage for single-precision floating point matrices. Several other SpMV implementations on GPUs have been described in [5, 6, 7] but do not achieve

Email address: dang@uni-mainz.de, bertil.schmidt@uni-mainz.de (Hoang-Vu Dang, Bertil Schmidt)

doi:10.1016/j.procs.2012.04.007

considerable performance compared to Cusp. Accelerating SpMV has a significant impact on computational science especially those applications that requires large scale iterative sparse linear solvers [8, 9]. They are also important in other fields such as graph traversal [10] or cryptography [11].

Performance evaluations reported in [1, 3, 4] are based on relatively small sparse matrices, i.e. matrices of size up to 1 million rows (columns) and less than 12 million non-zero coefficients. The irregular access pattern to the GPU memory can largely affect the performance of GPU implementations when evaluating larger matrices, while other architectures such as CPUs with much larger caches can moderate this effect. To overcome this challenge, we introduce a new format for SpMV named Sliced Coordinate List (SCOO) format. The proposed format has evolved from our previous work on SpMV for matrices derived from the General Number Field Sieve algorithm for factoring very large integers [11]. We extend the method of SpMV over $GF(2)$ to make it capable for general matrices over real numbers. Major modifications include generalizing three different formats: Small, Medium and Large Sliced COO to one format with different possible slice sizes, introducing a faster algorithm for matrix partitioning, and relaxing the memory requirements for improved performance. Our current implementation is limited to single-precision floating point matrices on CUDA-enabled GPUs with computing capability of 2.x such as Fermi. The idea however can be applied to other similar architectures with only modest modifications.

This paper is organized as follows. Section 2 provides background about the SpMV operation, important features of Fermi GPUs and the Cusp COO implementation. Section 3 describes our SCOO format and the corresponding CUDA algorithm. We compare our method to existing formats of Cusp and to an Intel Sandy-Bridge CPU using the publicly available Intel Math Kernel Library (MKL) software [12] in Section 4. Finally, Section 5 concludes the paper.

2. Background

2.1. SpMV

Let B denote the sparse input $d \times d$ matrix, x denote the dense $d \times 1$ input vector, y denote the dense $1 \times d$ output vector and γ denote the total number of non-zero entries of B. The *weight* of a row is the number of non-zeros in that row. The SpMV operation is defined as $y = B \cdot x$. Without loss of generality, we assume the matrix is square (non-square matrices can be padded with zeros). The input and output vector are stored in memory as two arrays of d elements. Algorithm 1 is a sequential version of SpMV assuming matrix B is stored in a 2D array. Obviously, the algorithm changes when the represenation of the matrix is different. The changes affect the order in which non-zeros are read and in which elements of x are accessed, thus have a large impact on GPU performance where memory access patterns are crucial.

Algorithm 1 Sequential SpMV

```
Input: B: d × d sparse matrix
  x: input vector
Output: y ← B · x
  for r ← 0 to d-1 do
    y[i] ← 0
    for each non-zero at row r, column c do
      y[i] ← y[i] + B[r][c] × x[c]
    end for
  end for
```

2.2. Important features of CUDA and the Fermi architecture

A CUDA device has multiple *Streaming Multiprocessors* (SMs), each can execute a *thread block* that was scheduled and assigned to it. Each thread is identified by their block and thread identification (*blockId, threadId*). The configurable variable *blockDim* defines the number of threads per block. Consecutive 32 threads form a *warp* which are executed in *SIMT* fashion (Single Instruction Multiple Threads [13, 14]). Hence, threads in a warp should avoid taking different paths of execution as much as possible to prevent *thread divergence*.

There are different types of memory in a CUDA device. The *global memory* is the slowest, however, largest in size and visible to all threads spawned by the device. Thus, the CUDA implementation of SpMV usually stores both matrix and vectors in global memory. Memory accesses to B and y can usually be coalesced, but memory accesses to x are random and non-coalesed, thus having higher latency. *Texture memory* can be used in that case to take advantage of the *texture cache*. *Shared memory* is a low latency memory but is small and only visible to threads in a block. Thus, it should only be used for intermediate results. The shared memory is divided into *banks* and when accessing shared memory *bank conflicts* should be avoided, otherwise the access will be serialized.

A Fermi GPU with compute capability of 2.x improves the performance for caching and atomic operation compared to previous generations. They also have more shared memory, i.e. 48KB compared to 16KB on lower compute capability devices. Our method exploits these improvements and we refer interested reader to a complete list of differences between various compute capabilities in [14].

2.3. Cusp implementation of the COO format

The COO format for storing B explicitly stores both row and column index of every non-zero entry. The Cusp library implementation stores non-zeros in row-major order, ensuring entries of the same row are contiguously stored. Implementing SpMV on CUDA with this storage format requires doing atomic updates to the y vector from parallel threads, which reduces performance. To overcome this bottleneck, Cusp uses parallel segmented reduction (S_REDUCTION) [1] on shared memory before writing to y. However, because shared memories are only visible to threads within the same block, results from different blocks still need to be combined in the global memory forming the output vector.

Algorithm 2 shows CUDA pseudocode for SpMV using the COO format. A matrix stored in COO format consists of three arrays of size γ each: `c_index`, `r_index`, `value` to store the column index, row index and value of each non-zero entry respectively. As observed from the algorithm, each element of x is accessed multiple times in a random order. `c_index` and `r_index` elements are accessed only once in a coaleased manner. Line 6 of Algorithm 2 shows the first S_REDUCTION for results in each thread block and Line 9 shows the second S_REDUCTION for results of the first reduction stored in the global memory. Cusp also optimizes the second S_REDUCTION by loading elements of `global` into shared memory for a faster reduction.

Algorithm 2 CUDA algorithm for SpMV on COO format

Input: $d \times d$ matrix B, stored in COO format as:
- (`c_index`, `r_index`, `value`)

CUDA-specific variable to identify a thread, number of threads per block and number of blocks launched
- (`threadId, blockId, blockDim, gridDim`)

`x`: input vector

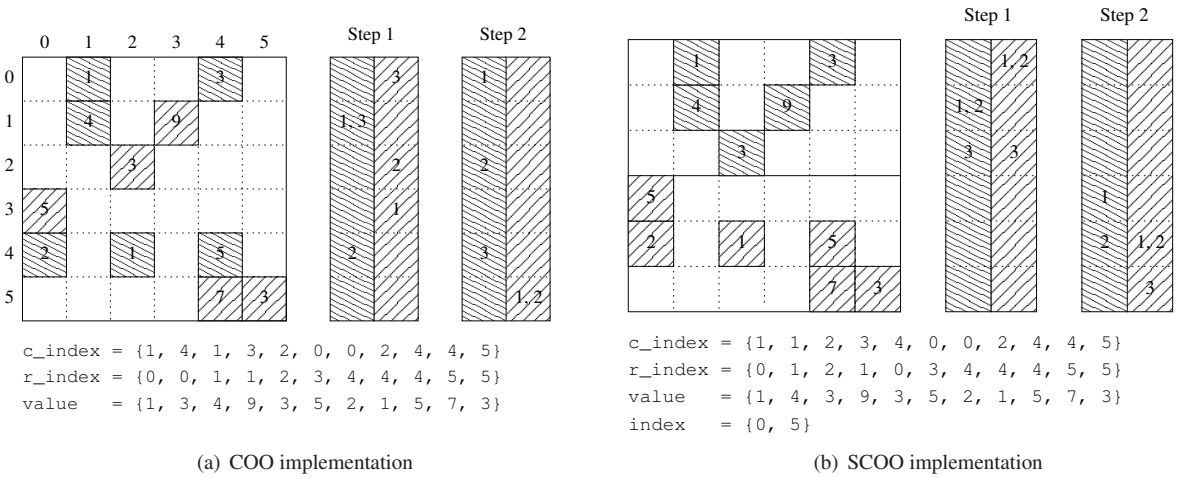
Output: $y \leftarrow B \cdot x$

```
1: i ← blockDim × blockId + threadId
2: init a global memory array global of size gridDim for intermediate results
3: init a shared memory array shared of size blockDim for intermediate results
4: while i < γ do
5:   shared[i] ← value[i] × x[c_index[i]]
6:   global ← S_REDUCTION_1(shared, r_index)
7:   i ← i + blockDim × gridDim
8: end while
9: y ← S_REDUCTION_2(global, r_index)
```

3. SpMV on a GPU using the SCOO format

3.1. SCOO format

We first sort the matrix rows by their weights using Quicksort. Similar to the standard COO format both row and column indices of each non-zero entry are stored. However, we sort them by column index so that entries of the same



(a) COO implementation (b) SCOO implementation

Figure 1: Data representation and access pattern of a 6×6 matrix stored in (a) COO format, (b) SCOO implementation with slice size = 3. We assume *blockDim* = 3, and *threadId* ranging from 1 to 3. We also assume two active thread blocks in both cases. Thus, before reduction, the SpMV requires two steps (in a step each thread multiply one matrix element with one input vector element). The access pattern is shown in two rectangles illustrating the input vector element that each thread in each block has to read. For the matrix, only entries with non-zero value are shown.

column are stored contiguously. By moving entries with similar column index together in a non-decreasing order, we want to maximize regular accesses to the input vector which can benefit performance. Recall that the input vector is stored in the CUDA global memory and each element can be read multiple times. Accessing CUDA global memory is only fast if threads in a warp access consecutive locations and addresses are 32-bytes aligned. Thus, our approach is able to improve the coaleased reading of the input vector. In the case of non-coalesed reading, the memory addresses can be still close to each other facilitating more cache-friendly access patterns. Figure 1 illustrates the differences in accesss pattern between SCOO and COO.

In SCOO the row indices need to be accessed in random order. This implies an unpredictable access pattern when combining the intermediate results, hence S_REDUCTION as in Algorithm 2 is no longer possible. In order to overcome this problem, we divide the matrix into multiple slices of consecutive rows and only sort the non-zero entries locally in each slice. Hence, the value of row indices in a slice are within a managable range and we can fit the intermediate results in a faster memory for the reduction. On Fermi GPUs, we specifically use the 48KB per block shared memory for that purpose.

Let h denote the number of rows per slice, S denote the size in byte of shared memory per thread block ($S = 49152$ for Fermi) and b denote the size of each matrix value in byte ($b = 4$ for single-precision floating point). h must be a positive integer value of the form of $h = \frac{S}{2^i b}$, $\forall 0 \leq i \leq T$ where $T = \log_2$ *(maximum number of thread per block)*. For example in Fermi GPUs, single-precision floating point, $h \in \{12288, 6144, 3072, 1536, 768, 384, 192, 96, 48, 24, 12\}$ for $S = 49152, b = 4, T = 10$.

Additionally, the SCOO format requires an ordered list $[(H_1, R_1),(H_2, R_2),...,(H_K, R_K)]$ $(0 < K \leq T + 1)$ such that $\forall 1 \leq i < K, H_i < H_{i+1}, R_1 = 0 \leq R_i < R_{i+1} \leq R_K < d$. This indicates that from row R_i to row $R_{i+1} - 1$ of the preprocessed matrix the SCOO format is built with the slice size H_i. Figure 2 illustrates the SCOO format of a sparse matrix. We describe how to determine the values (H_i, R_i) in Section 3.3.

Let $B^{r_1}_{r_2}$ denote the submatrix of the sorted input matrix B acquired by taking the rows from r_1 to $r_2 - 1$ $(0 \leq r_1 < r_2 \leq d)$. SpMV using the SCOO format requires executing a seperate CUDA SpMV kernel for each submatrix $B^{R_i}_{R_{i+1}}$.

3.2. CUDA kernel for a fixed slice size

We present a CUDA algorithm for SpMV using the SCOO format given a slice size h in Algorithm 3. We launch as many thread blocks as the number of slices and assign one thread block to work on one SCOO slice, thus the slice

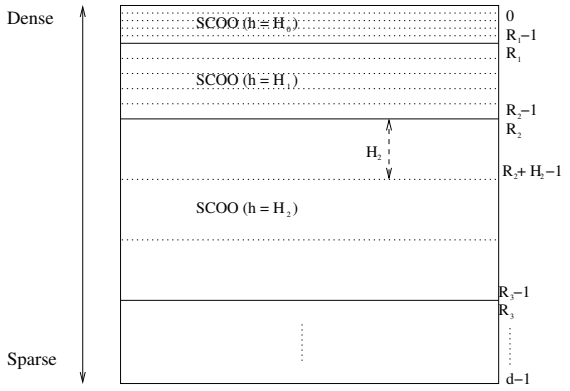


Figure 2: SCOO format of a sparse matrix with rows sorted by their weights in non-increasing order.

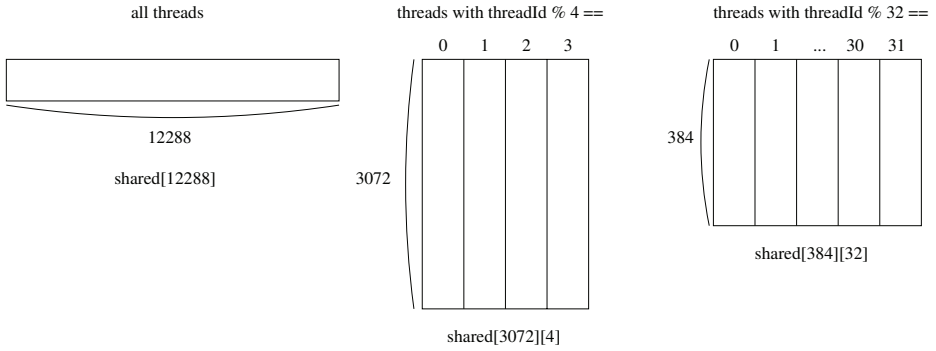


Figure 3: Shared memory partitioning for Fermi GPUs using $b = 4$, $h = 12288, 3072, 384$. Threads in a block should only access the memory lane with the given condition.

identification is equal to `blockId`. A matrix stored in SCOO format using k slices of size h each consists of four arrays:

- `c_index`, `r_index`, `value` of size γ each to store the column indices, the row indices and the values of γ non-zero entries respectively.
- `index` of size k to store the pointer to the start of each slice in `c_index`, `r_index`, `value`.

Comparing Algorithm 3 to Algorithm 2, the major difference is the use of *atomicAdd*. An atomic operation is required to synchronize the access to each shared memory entry in the event that more than one parallel thread is trying to access the same memory address. In general, decreasing h increases the chance two coefficients having the same row index are processed at the same time causing serialization. In serialization, a thread has to wait for another thread which causes all other threads in the same warp to stall while waiting for it. This effect is similar to thread divergence and thus reduces the performance.

To reduce such events, we allow multiple lanes in the shared memory for updating the intermediate results of a single row. The number of lanes in Algorithm 3 is L. Figure 3.2 shows the presentation of shared memory for different value of h. Reducing h increases L and thus more shared memory lanes are available.

Lines 12 to 16 of Algorithm 3 write the results in a coalesed manner to the global memory where the result vector y is stored. When `blockDim` is larger than L, the parallel reduction in Line 11 can be improved by performing reduction on multiple rows in parallel. In that case `shared[i][i mod L]` can be used instead of `shared[i][0]` to reduce bank conflicts.

Algorithm 3 Parallel algorithm for SpMV using SCOO format slice size of h on CUDA devices

Input: $d \times d$ matrix B, stored in SCOO format with slice size h as:
- (c_index, r_index, value, index)
CUDA-specific variable to identify a thread and number of thread per block:
- (threadId, blockId, blockDim)
x: input vector

Output: y $\leftarrow B \cdot$ x: output vector

```
1: i ← index[blockId] + threadId
2: end ← index[blockId+1]
3: L ← S/hb
4: lane ← threadId mod L
5: init shared memory array shared[h][L]
6: while i < end do
7:   r ← r_index[i] - blockId × h
8:   s[r][lane] ← atomicAdd( s[r][lane] , value[i] × x[c_index[i]] )
9:   i ← i + blockDim
10: end while
11: parallel reduction shared[i][0] ← Σ_{j=0}^{L-1} shared[i][j], ∀i = 0..h − 1
12: i ← threadId
13: while i < h do
14:   y[blockId × h + i] ← shared[i][0]
15:   i ← i + blockDim
16: end while
```

3.3. Matrix partitioning

Let M denote the number of SMs of a given GPU. In order to avoid idling SMs we always select the same slice size h for a set of M consecutive slices. As a result, the number of rows per slice in the SCOO format is a multiple of $M \times h$. Within slice groups that have an unbalanced number of non-zeros, we further re-order the rows to achieve better load balancing.

To partition the matrix, we can use the same method as the cut-off point determination in [11]. For this method, we execute the kernel with different slice size multiple times from a specific row to determine the value that gives the best performance in terms of Giga floating-point operation per second (Gflop/s). The method is slow since the sub-matrices have to be transferred multiple times to GPU memory. Hence, we propose a faster heuristic to partition the matrix which uses following notations:

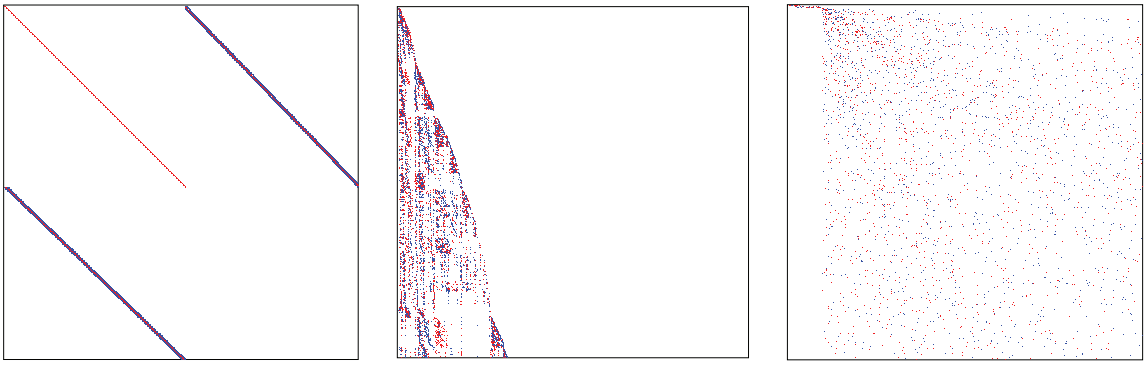
- $\epsilon = \frac{S}{b}$: total number of shared memory entry per thead block.
- $\theta = \frac{\gamma}{d}$: avarage number of non-zeros per row of the input matrix.
- $\gamma(B^{r_1}_{r_2})$: the number of non-zeros in the submatrix $B^{r_1}_{r_2}$.
- $\delta(B^{r_1}_{r_2})$: the avarage number of accesses per shared memory entry for $B^{r_1}_{r_2}$ stored in SCOO format with a fixed slice size.

Recall that the result is an ordered list $[(H_1, R_1),(H_2, R_2),...,(H_K, R_K)]$ $(0 < K \leq T + 1)$ such that $0 = R_1 \leq R_i < R_{i+1} \leq R_K < d, H_i < H_{i+1}, \forall i \in \{1, 2, ..., K\}$. Our algorithm consists of two steps:

- **Step 1**: Find an ordered list $[(H'_1, R'_1), (H'_2, R'_2), ..(H'_{K'}, R'_{K'})]$ $(0 < K' < \frac{d}{M \times \text{ minimum valid slice size}})$ such that:
  - $0 = R'_1 \leq R'_i < R'_{i+1} \leq R'_{K'} < d, H'_i \leq H'_{i+1}, \forall i \in \{1, 2, ..K'\}$
  - $\theta \leq \delta(B^{R'_i}_{R'_{i+1}}) \leq 2\theta, \forall i \in \{1, 2, ..., K' - 1\}$
  - $R'_{i+1} = R'_i + MH'_i, \forall i \in \{1, 2, ..K' - 1\}$

Table 1: Overview of hardware used in the experiments

Hardware	C2075	GTX-580	Core-i7 2700K
# Cores	448	512	4
Clock speed (Ghz)	1.15	1.57	3.5
Memory type	GDDR5	GDDR5	DDR3-1600
Memory size (GB)	6	3	16
Max Memory bandwidth (GB/s)	144	192	21



(a) nlpkkt120 (b) Relat9 - first 10000 rows (c) GL7d19 - first 500 rows and columns

Figure 4: Visualization of *nlpkkt120 Relat9* and *GL7d19* matrix using MatView [15]. The white color part represents zero entries, red and blue part represent non-zero entries with value larger or smaller than zero respectively

– H'_i *is a valid slice size*, $\forall i \in \{1, 2, ..., K'\}$

From a matrix row $r = R'_t$, we want to make sure that the slice size $h = H'_t$ of the SCOO format using to form the submatrix B^r_{r+Mh} satisfies the condition $\theta \leq \delta(B^r_{r+Mh}) \leq 2\theta$. Because each slice has a similar number of non-zeros and will be assigned to exactly one thread block, $\delta(B^r_{r+Mh})$ can be estimated as $\frac{\gamma(B^r_{r+Mh})}{M\epsilon}$. Recall that since the matrix rows are sorted, we can always find a valid slice size because doubling h does not double δ except for the two extreme cases ($h = 12$ and $h = 12288$ for Fermi) in which we select the smallest and largest value as a result.

- **Step 2**: Merge any consecutive elements of the list from j to j' ($1 \leq j < j' \leq K'$) such that $H'_j = H'_{j+1} = .. = H'_{j'}$ to generate the final result.

The aim of our new algorithm is to ensure that each matrix slice has a similar number of shared memory accesses. The complexity of this algorithm is only $O(d)$ since $\gamma(B^r_{r+Mh})$ can be calculated in $O(1)$ using a precomputed prefix sum of matrix row weights.

3.4. Memory Usage

The Cusp COO implementation for single-precision floating point values requires 4 bytes per element for each of the three arrays: `c_index`, `r_index`, `value`. Our SCOO format stores the matrix in slice-major order and therefore we only need only 2 bytes per element for `r_index`, and effectively reduce the overall memory required for storing the matrix by 2 bytes per non-zero compared to the COO format.

Table 2: Overview of sparse matrices used for performance evaluation

Name	row	column	non-zero(nz)	nz/row	Description
GL7d19	1,911,130	1,955,309	37,322,725	19,53	combinatorial problem
relat9	12,360,060	549,336	38,955,420	3,15	combinatorial problem
wikipedia-20070206	3,566,907	3,566,907	45,030,389	12,62	directed graph
wb-edu	9,845,725	9,845,725	57,156,537	5,81	directed graph
road_usa	23,947,347	23,947,347	57,708,624	2,41	undirected graph
hugebubbles-00010	19,458,087	19,458,087	58,359,528	3,00	undirected graph
circuit5M	5,558,326	5,558,326	59,524,291	10,71	circuit simulation
nlpkkt120	3,542,400	3,542,400	95,117,792	26,85	optimization problem
cage15	5,154,859	5,154,859	99,199,551	19,24	directed weighted
kron_g500-logn21	2,097,152	2,097,152	182,082,942	86,82	undirected multigraph
indochina-2004	7,414,866	7,414,866	194,109,311	26,18	directed graph
nlpkkt160	8,345,600	8,345,600	225,422,112	27,01	optimization problem
rgg_n_2_24_s0	16,777,216	16,777,216	265,114,400	15,80	undirected random
uk-2002	18,520,486	18,520,486	298,113,762	16,10	directed graph

4. Experimental Result

4.1. Experimental Setup

In our experiments, we compare the performance of our SCOO format to available SpMV implementations on both GPU and CPU. The set of selected test matrices are collected from the University of Florida Sparse Matrix Collection [16]. We have chosen the biggest matrices from different areas that with their corresponding input and output vector can still fit into the 6GB global memory of a C2075 GPU. Table 2 gives an overview of those matrices.

Table 1 gives an overview of the GPUs and the CPU workstation used for performance evaluation. The performance is measured in terms of Gflop/s. Measured GPU performance does neither include PCIe data transfers nor matrix preprocessing. These are realistic assumption since SpMV applications usually consist of a large number of iterations where the sparse matrix is iteratively multiplied by the input/output vectors.

4.2. Performance comparison to existing GPU formats

We compare the SCOO format to the CSR, COO and HYB format of Cusp. Other Cusp formats are not able to run on the large tested matrices that we selected. Figure 5 shows the results. The SCOO format achieves a stable performance for different matrices. In most cases a performance of over 10 Gflop/s can be sustained. For some highly unstructure matrices such as *GL7d19*, *wikipedia-20070206*, *rgg_n_2_24_s0* and *kron_g500-logn21* SCOO achieves high speedups ranging from 3 to 6 compared to the best performancing Cusp format. Figure 4(c) visualizes the unstructured *GL7d19* sparse matrix.

For most matrices, HYB produces the best performance among all the tested Cusp formats. HYB is able to outperform SCOO only for two matrices: *nlpkkt120* and *nlpkkt160*. Both matrices have a similar structure. They consist of consecutive rows that have a very similar number of non-zero coefficients which is suitable to be stored in the ELL section of the HYB format. Moreover the non-zeros are close to each other facilitating coaleasing and cache-friendly access patterns by nature. We show the visualization of the sparse matrix *nlpkkt120* in Figure 4(a). SCOO is able to outperform COO and CSR for all tested matrices.

We show in Figure 4(b) the visualisation of matrix *Relat9*, in which we observe some high density regions of non-zeros. However the matrix is still generally unstructured, thus SCOO is able to achieve about 2 times speed up compared to HYB which is the best among tested Cusp formats in this case.

The memory usage of each format for each matrix is shown in Figure 6. For large matrices, SCOO consumes more memory than CSR but less than COO and HYB.

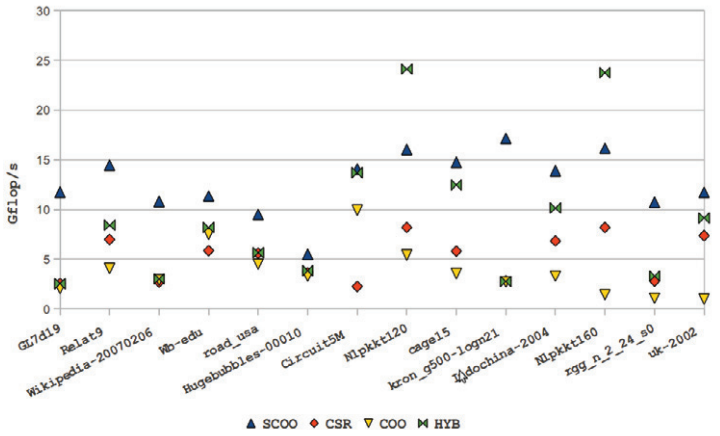


Figure 5: Single-precision performance in terms of Gflop/s of SCOO and other GPU formats for each test matrix on a Fermi Tesla C2075 (ECC disabled). Gflop/s values are based on the assumption of two flops per non-zero entry of the matrix [1, 3]

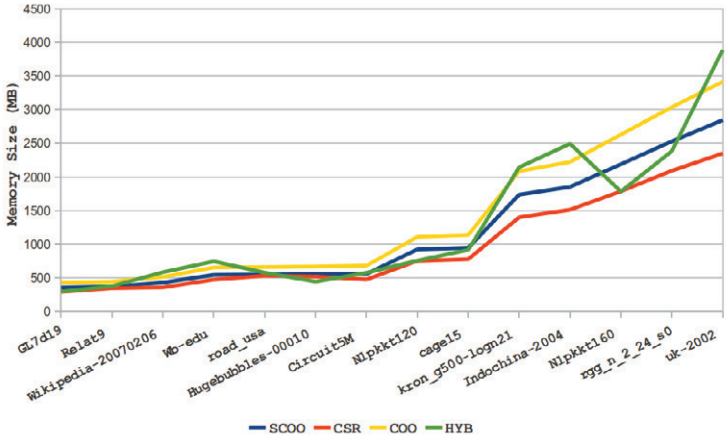


Figure 6: GPU memory required to store the matrix in different formats for matrices from smallest to largest

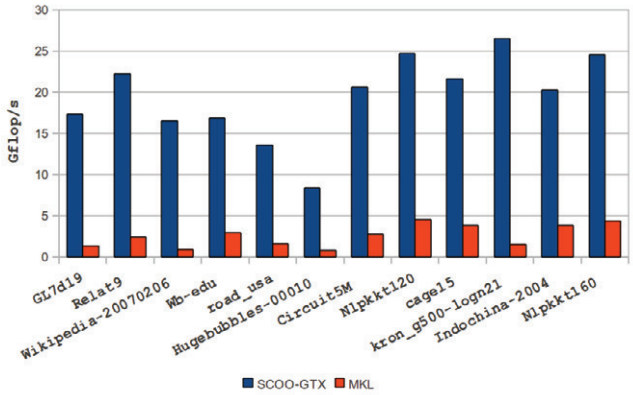


Figure 7: Single-precision performance in terms of Gflop/s of the SCOO on a GTX-580 and a CPU implementation using MKL performed on a Core-i7 2700K using 8 threads.

4.3. Performance comparison to a CPU implementation

We use the Intel MKL library to compare the performance to an optimized CPU implementation. MKL SpMV receives the input matrices in CSR format. The results are shown in Figure 7. Using a GTX-580, we achieve speedups ranging between 5.5 and 18 over MKL on a 4-core CPU using 8 threads. Also note that the SCOO performance on a GTX-580 is around 1.5 times faster than on the C2075 due to the increased memory bandwidth and clock speed. The storage requirement for the *rgg_n_2_24_s0* and *uk-2002* matrices and associated input/output vectors slightly exceeds the 3 GB global memory of the GTX-580 and thus are not included in Figure 7.

5. Conclusion

In this paper, we have described a new method for performing SpMV on a CUDA-enable GPU. The SCOO format and its corresponding CUDA algorithm have been presented. Our evaluation shows that SCOO can significantly improve SpMV performance compared to existing formats of the Cusp library for large unstructured matrices. We have further presented an efficient heuristic to convert a given sparse matrix into SCOO format. Our future work includes extending our proposed method for single-precision to double-precision floating point matrices. Another direction would be combining the SCOO format with the Sliced Ellpack format to form a new hybrid format similar to the HYB format of Cusp library in order to improve the performance even further.

References

[1] N. Bell, M. Garland, Implementing sparse matrix-vector multiplication on throughput-oriented processors, in: SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, ACM, New York, NY, USA, 2009, pp. 1–11. doi:http://doi.acm.org/10.1145/1654059.1654078.

[2] N. Bell, M. Garland, Cusp: Generic Parallel Algorithms for Sparse Matrix and Graph Computations, `http://cusp-library.googlecode.com`, version 0.2.0 (2010).

[3] J. W. Choi, A. Singh, R. W. Vuduc, Model-driven autotuning of sparse matrix-vector multiply on GPUs, SIGPLAN Not. 45 (2010) 115–126.

[4] A. Monakov, A. Lokhmotov, A. Avetisyan, Automatically Tuning Sparse Matrix-Vector Multiplication for GPU Architectures, in: HiPEAC, 2010, pp. 111–125.

[5] M. M. Baskaran, R. Bordawekar, Optimizing sparse matrix-vector multiplication on gpus, Tech. rep., IBM TJ Watson Research Center (2008).

[6] L. Buatois, G. Caumon, B. Lvy, Concurrent number cruncher: An efficient sparse linear solver on the gpu, in: High Performance Computation Conference (HPCC), Springer Lecture Note s in Computer Sciences, 2007.

[7] F. Vazquez, G. Ortega, J. J. Fernandez, E. M. Garzon, Improving the performance of the sparse matrix vector product with gpus, in: Proceedings of the 2010 10th IEEE International Conference on Computer and Information Technology, CIT '10, IEEE Computer Society, Washington, DC, USA, 2010, pp. 1146–1151.

[8] L. Buatois, G. Caumon, B. Lvy, Concurrent number cruncher - a gpu implementation of a general sparse linear solver, International Journal of Parallel, Emergent and Distributed Systems.

[9] Y. Saad, Iterative Methods for Sparse Linear Systems, 2nd Edition, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2003.

[10] D. Merrill, M. Garland, A. Grimshaw, High performance and scalable gpu graph traversal, in: Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2012), ACM, 2012.

[11] B. Schmidt, H. Aribowo, H.-V. Dang, Iterative sparse matrix-vector multiplication for integer factorization on gpus, in: E. Jeannot, R. Namyst, J. Roman (Eds.), Euro-Par 2011, Vol. 6853 of Lecture Notes in Computer Science, Springer, 2011, pp. 413–424.

[12] Intel Coporation, Intel Math Kernel Library, `http://software.intel.com/en-us/articles/intel-mkl/`, version 10.3 (2012).

[13] J. Nickolls, I. Buck, M. Garland, K. Skadron, Scalable parallel programming with cuda, Queue 6 (2008) 40–53.

[14] NVIDIA Corporation, Cuda c programming guide, `http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf` (2011).

[15] J. Kohl, Matview: Scalable sparse matrix viewer, `http://www.csm.ornl.gov/~kohl/MatView/` (2008).

[16] I. S. Duff, R. G. Grimes, J. G. Lewis, Sparse matrix test problems, ACM Trans. Math. Softw. 15 (1989) 1–14.